

LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84

Il6r

no. 293-300

cop. 2



CENTRAL CIRCULATION AND BOOKSTACKS

The person borrowing this material is responsible for its renewal or return before the **Latest Date** stamped below. **You may be charged a minimum fee of \$75.00 for each non-returned or lost item.**

Theft, mutilation, or defacement of library materials can be causes for student disciplinary action. All materials owned by the University of Illinois Library are the property of the State of Illinois and are protected by Article 16B of Illinois Criminal Law and Procedure.

TO RENEW, CALL (217) 333-8400.

University of Illinois Library at Urbana-Champaign

JUN 28 1999

FEB 01 A.M.

When renewing by phone, write new due date
below previous due date.

L162



Digitized by the Internet Archive
in 2013

<http://archive.org/details/storageallocatio297mura>

262
0.297
Report No. 297

STORAGE ALLOCATION ALGORITHMS
IN THE TRANQUIL COMPILER

by

Yoichi Muraoka

January 13, 1969



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

卷之四

Report No. 297

STORAGE ALLOCATION ALGORITHMS
IN THE TRANQUIL COMPILER*

by

Yoichi Muraoka

January 13, 1969

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

* This work was supported in part by the Advanced Research Projects Agency as administered by the Rome Air Development Center under Contract No. US AF 30(602)4144 and submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, February, 1969.

ACKNOWLEDGEMENT

The author would like to express his most sincere gratitude to Professor Robert S. Northcote, Department of Computer Science of the University of Illinois, who graciously offered many suggestions and comments. Furthermore, without his criticism of the presentation, this thesis would probably be totally unreadable.

The author is also indebted to Professor David J. Kuck, who originated TRANQUIL and provided helpful criticism and suggestions throughout this work.

Thanks are also extended to Mrs. Patricia Stippes, who typed the manuscript in its final form.

ABSTRACT

TRANQUIL is a language for describing algorithms in terms of parallel constructs. Its compiler is now being implemented for the parallel array computer ILLIAC IV. This paper discusses a particular part of the implementation; namely, the problem of storage allocation for arrays.

2. 10. 1941 1. 10. 1941 2. 10. 1941 3. 10. 1941 4. 10. 1941 5. 10. 1941 6. 10. 1941 7. 10. 1941 8. 10. 1941 9. 10. 1941 10. 10. 1941 11. 10. 1941 12. 10. 1941 13. 10. 1941 14. 10. 1941 15. 10. 1941 16. 10. 1941 17. 10. 1941 18. 10. 1941 19. 10. 1941 20. 10. 1941 21. 10. 1941 22. 10. 1941 23. 10. 1941 24. 10. 1941 25. 10. 1941 26. 10. 1941 27. 10. 1941 28. 10. 1941 29. 10. 1941 30. 10. 1941 31. 10. 1941 32. 10. 1941 33. 10. 1941 34. 10. 1941 35. 10. 1941 36. 10. 1941 37. 10. 1941 38. 10. 1941 39. 10. 1941 40. 10. 1941 41. 10. 1941 42. 10. 1941 43. 10. 1941 44. 10. 1941 45. 10. 1941 46. 10. 1941 47. 10. 1941 48. 10. 1941 49. 10. 1941 50. 10. 1941 51. 10. 1941 52. 10. 1941 53. 10. 1941 54. 10. 1941 55. 10. 1941 56. 10. 1941 57. 10. 1941 58. 10. 1941 59. 10. 1941 60. 10. 1941 61. 10. 1941 62. 10. 1941 63. 10. 1941 64. 10. 1941 65. 10. 1941 66. 10. 1941 67. 10. 1941 68. 10. 1941 69. 10. 1941 70. 10. 1941 71. 10. 1941 72. 10. 1941 73. 10. 1941 74. 10. 1941 75. 10. 1941 76. 10. 1941 77. 10. 1941 78. 10. 1941 79. 10. 1941 80. 10. 1941 81. 10. 1941 82. 10. 1941 83. 10. 1941 84. 10. 1941 85. 10. 1941 86. 10. 1941 87. 10. 1941 88. 10. 1941 89. 10. 1941 90. 10. 1941 91. 10. 1941 92. 10. 1941 93. 10. 1941 94. 10. 1941 95. 10. 1941 96. 10. 1941 97. 10. 1941 98. 10. 1941 99. 10. 1941 100. 10. 1941

TABLE OF CONTENTS

	Page
1. INTRODUCTION	1
2. DATA STRUCTURES IN TRANQUIL	3
2.1 <u>Data Declarations</u>	3
2.2 <u>Mapping Functions</u>	3
3. IMPLEMENTATION	9
3.1 <u>Introduction</u>	9
3.2 <u>Array Partitioning</u>	10
3.3 <u>Address Calculation</u>	24
3.4 <u>Storage Allocation</u>	26
4. FURTHER DISCUSSION	29
5. CONCLUSION	31
APPENDIX	
A. SYNTAX AND SEMANTICS SPECIFICATION OF TRANQUIL DECLARATIONS.	32
1. <u>Declarations</u>	32
2. <u>Variable Declaration</u>	33
3. <u>Array Declaration</u>	34
4. <u>PEM Reserve Declaration</u>	39
5. <u>PEM Assignment Declaration</u>	40
B. TABLES	43

	Page
C. ARRAY PARTITIONING AND PACKING FLOWCHARTS.	45
D. STORAGE ALLOCATION PACKAGES.	50
LIST OF REFERENCES	55

THE UNIVERSITY OF CHICAGO PRESS
1215 EAST 58TH STREET
CHICAGO, ILL. 60637
U.S.A. AND CANADA
OTHER COUNTRIES
BY POSTAL ORDER
OR BY DIRECT DEPOSIT
WITH THE NATIONAL TRADING
STAMP AUTHORITY

LIST OF FIGURES

Figure	Page
1. The Standard Storage Schemes.	6
2. Partitioning for Array A [1:3, 1:300, 1:300].	11
3. Block Packing for Array A [1:3, 1:300, 1:300]	18
4. Use of PE Memory for the Blocks Shown in Figures 2 and 3	19
5. BASETB Format 1	20
6. BASETB Format 2	20
7. Examples of BASETB Entries.	21
8. One Subarray Generated from Array A [#1:10, ##1:10, *1:3, **1:5].	22
9. SLIST	27
 A1. Subarrays for an Array A [#5, ##4, *32, **64]	 37
B1. Entries and Linkage of Tables for A[1:M ₁ , 1:M ₂ , ..., 1:M _n].	44
C1. Pass 2 Program Block Entry and Block Exit Flowcharts for Array Declarations.	46
C2. Array Partitioning Flowchart.	48
C3. Residual Block Packing Flowchart.	49
D1. Table and List Entry Formats.	52
D2. Example of an Entry for I4MEMORY.	53
D3. Example of an Entry for VLIST	54

THE UNIVERSITY OF CHICAGO
LIBRARY
1100 EAST 58TH STREET
CHICAGO, ILL. 60637
U.S.A.

1. INTRODUCTION

Familiarity with the structure of the ILLIAC IV computer [1] and the TRANQUIL language [2] will, in general, be assumed throughout this paper. However a few of the characteristics of ILLIAC IV which are important to the development of this paper will now be given.

The most important feature of ILLIAC IV is that many simple identical processing elements (PEs) obey instructions which are decoded by one common control unit (CU). Each PE can receive common data which is broadcast from its CU, but it also has access to data in its own 2K word memory (PEM). Thus, although executing identical decoded control signals from a CU, every PE can operate upon different data.

Each PE has the option of playing either an active or a passive role during an instruction cycle on the basis of its own state, which is determined by mode control. Also, each PE is connected to its four nearest neighbors, thus permitting the routing of data from one PE to another.

If ILLIAC IV is to be used efficiently, it is essential that data be stored "evenly" in PEM's so that, on receiving control signals from a CU, as many PE's as possible can operate on their own data, i.e., the data which are stored in their own PEM. Further, if it should be necessary for some PE to use data in another PEM, the routing distance should be as small as possible.

TRANQUIL, as a language, has been designed to assist in the programming of primarily algebraic numerical computations. Among many existing computer languages, ALGOL, FORTRAN, PL1, and APL [3] also fall into this category. The prime concern here is to decide what kind of data types (information structures other than simple variables) should be included in the language. According to Knuth [4], information structures can be categorized as follows:

- (i) linear list
- (ii) tree
- (iii) multilinked structure.

In the languages mentioned above, however, not all of these structures are provided explicitly. Arrays are usually the only information structure which is provided (although multilinked structures are allowed in PL1). Other information structures such as the tree structure are implemented by utilizing arrays [3]. TRANQUIL also has arrays as the only structured data type in the language. The other types of data structures were not included because ILLIAC IV has been designed as an array machine and computations on arrays are its primary function.

In the following chapters, the problem of storage allocation in the TRANQUIL compiler, namely how to specify mapping functions for arrays and how the TRANQUIL compiler handles them, are discussed.

2. DATA STRUCTURES IN TRANQUIL

2.1 Data Declarations

The data structures which are recognized in TRANQUIL are simple variables, arrays and sets. (For further discussion on sets, the reader is referred to [8].) All data structures, as well as labels, switches, and procedures must be declared in some block head as in ALGOL. The data type attributes are INTEGER, REAL, COMPLEX and BOOLEAN. Certain precision attributes also may be specified. Array declarations must include attributes which specify type (as above) and storage scheme (i.e., mapping function), in addition to size specifications which follow the same format as those in PL1; e.g.,

REAL SKewed ARRAY A [1:50, 1:50].

As in ALGOL, the size of an array may be specified dynamically in inner blocks.

A complete syntax specification and informal semantics for declarations is given in Appendix A.

2.2 Mapping Functions

As was implicitly mentioned in the introduction, the efficiency of a program is highly dependent on the choice of mapping functions for arrays. Hence, TRANQUIL should provide users with a means of specifying mapping functions which are both simple to use and yet capable of specifying quite complicated storage schemes.

Several conventions for the specification of mapping functions are provided in TRANQUIL. These include both predefined standard storage schemes and a mechanism to enable users to specify their own scheme. Before investigating these schemes we discuss the partitioning of arrays, and the use of the PE memories as a two-dimensional array.

ILLIAC IV memory may be regarded as a 2048×256 array (number of words in a PEM) \times (number of PEM's). Hence, it is desirable to reduce all n -dimensional arrays ($n > 2$) to a set of 2-dimensional subarrays. Rows of these subarrays are usually stored in a row of ILLIAC IV memory: i.e., across PEM's. As an example, if the 3-dimensional array A is declared:

```
REAL  STRAIGHT  ARRAY A [1:3, 1:100, 1:200],
```

then the compiler will treat this as a set of three subarrays each of size 100×200 . Note that the size of the subarrays to be formed is, in general, determined by the last two dimensions.

In some cases it may be desirable to change this rule; i.e., form subarrays from dimensions other than the last two. To satisfy this requirement the following convention has been adopted. One asterisk placed in front of any dimension size specification in an array declaration forces that dimension to be stored in a column (i.e., one PEM) of ILLIAC IV memory, while two asterisks indicate row storage (i.e., across PEM's). Thus,

```
REAL  STRAIGHT  ARRAY A [**1:100, *1:200, 1:3]
```

forces the compiler to form three subarrays of size 200×100 .

Furthermore,

```
REAL  ARRAY  A [*1:50]
```

causes the vector A to be stored in one PEM.

PEM

	1	2	3	4	5	6	7	8
1	(1,1)	(1,2)	(1,3)	(1,8)
2	(2,1)	(2,2)	(2,3)	(2,8)
3
4
5
6
7
8	(8,1)	(8,2)	(8,8)

STRAIGHT

PEM

	1	2	3	4	5	6	7	8
1	(1,1)	(1,2)	(1,3)	(1,8)
2	(2,8)	(2,1)	(2,2)	(2,3)
3	.	(3,8)	(3,1)	(3,2)	(3,3)	(3,4)	.	.
4	.	.	.	(4,1)
5	(5,1)	.	.	.
6
7
8	(8,2)	(8,8)	(8,1)

SKEWED

(1,32) PE244	(2,32) PE242	.	(16,32) PE256	(17,32) PE1	..	(32,32) PE16
:	:	:	:	:	:	:
:	:	:	:	:	:	:
(1,17) PE1	.	.	(16,17) PE16	(17,17) PE17	.	(32,17) PE32
(1,16) PE241	(2,16) PE242	..	(16,16) PE256	(17,16) PE1	..	(32,16) PE16
:	:	:	:	:	:	:
(1,2) PE17	(2,2) PE18	..	(16,2) PE32	.	.	.
(1,1) PE1	(2,1) PE2	..	(16,1) PE16	(17,1) PE17	..	(32,1) PE32

Here 256PEM's (instead of 16) are used to store a 32 X 32 mesh point in the CHECKER scheme.

PEM

	1	2	3	4	5	6	7	8
1	(1,1)	(2,1)	(3,1)	(8,1)
2	(8,2)	(1,2)	(2,2)
3	.	(8,3)	(1,3)	(2,3)
4	.	.	(8,4)	(1,4)	(2,4)	.	.	.

An 8 X 4 matrix is stored in SKEWED PACKED scheme.

Figure 1. The Standard Storage Schemes

The standard, more commonly used, methods of storage are:

STRAIGHT, SKEWED and CHECKER.

These schemes are illustrated by the examples in Figure 1 where an 8×8 matrix A is stored in eight PEM's of ILLIAC IV. STRAIGHT is the simplest storage form for a matrix and leads to the simplest program, but the drawback is that a column is contained entirely in one PEM, thus prohibiting simultaneous access to all elements of a column. SKEWED allocation, on the other hand, distributes columns, as well as rows, across PEM's allowing access to an entire column in parallel. Since rows and columns are equally accessible in this scheme, matrices can be PACKED by appropriate transposition, thus minimizing wastage of memory space, as shown in Figure 1. The CHECKER allocation scheme has been developed specially for storing mesh type data for elliptic partial differential equations. This scheme allows each PE fast access to the four nearest neighboring mesh points. Further discussion on storage schemes for matrix type operations is found in [5]. The CHECKER scheme is discussed in [6]. Mapping functions are applied to the aforementioned subarrays. Thus, for example, skewing is done on each of three subarrays of size 200×100 belonging to the array A which is declared

REAL SKEWED ARRAY A [**1:100, *1:200, 1:3].

It is feasible to include new mapping functions in this list, but it

is anticipated that most users needs will be satisfied by this set of standard mapping functions.

Finally, the user who wishes to specify his own mapping function may make use of a PE memory assignment statement. For example:

```
PEMEMORY  PB [1:10, 1:256];
PEM FOR  (I, J) SIM ([1, 2, ..., 10] × [1, 2, ..., 256]) DO
      PB [I, J] ← B [I, MOD (256, I + J - 1)];
REAL ARRAY  (PB)  B[1:10, 1:256];
```

establishes virtual space of size 10×256 in PE memory, and then stores a 10×256 array B there in skewed form. Thus, instead of making up the aforementioned subarrays out of an array declaration, space reserved in PE memory may be used. In the program, the programmer refers to an element of memory space via the assigned array name B and its subscripts, as usual.

It should be noted that storage mapping functions can not be specified dynamically. Should remapping of data be required, an explicit assignment statement may be used; e.g., to change the data in an array A from skewed to straight storage an assignment statement

$$B \leftarrow A$$

is used, where B has been declared to be a straight array.

3. IMPLEMENTATION

3.1 Introduction

The TRANQUIL compiler currently has two passes. In pass 1, on recognizing declarations, the compiler enters the necessary descriptive information; e.g., in the case of array declarations, the attributes, type of mapping function, and size and dimension information for the array, into a table (IDTAB). Segmentation of data, i.e., partitioning of arrays, and storage allocation are taken care of in pass 2. Descriptions and formats of the tables used are given in Appendix B.

There are many computational problems the programs for which require more working storage than is available. This is particularly true for programs which will be run on ILLIAC IV, because the size of each PEM is relatively small compared with the computational speed of a PE. For example, multiplication of two 256×256 matrices takes only 70 msec on ILLIAC IV, but almost half the memory is needed to accomodate three of these arrays. Thus a strategy for segmenting arrays, and controlling the overlaying of segments, must be devised for the TRANQUIL compiler. The segmentation of object programs will not be considered here. Our main concern is how to deal with large arrays.

3.2 Array Partitioning

According to Randell and Kuehner [7], two characteristics believed to be most useful for revealing the functional capability and underlying mechanisms of current hardware-assisted dynamic storage allocation systems are related to the concepts of name space and predictive information. A discussion of these characteristics will now be given.

(i) Name Space

On conventional computers, memory is always treated as a linear space. Array elements must be stored as a linear array or vector; e.g., matrices are stored in order by rows, and a row of an array frequently forms a segment [7]. A typical size for a segment is 1024 words (B5500), and this limitation is reflected in the fact that the maximum size vector that can be declared is 1024 words.

ILLIAC IV memory, however, may be regarded as a two-dimensional space, i.e. an array, with 2048 rows (number of words in a PEM) \times 256 columns (number of PEM's). This suggests the use of a two-dimensional segment, to be referred to as a block. Explicitly, all arrays will be partitioned into two dimensional-blocks, the maximum size of which is 256×256 words. This partitioning is done on the subarrays which were mentioned in Chapter 2.

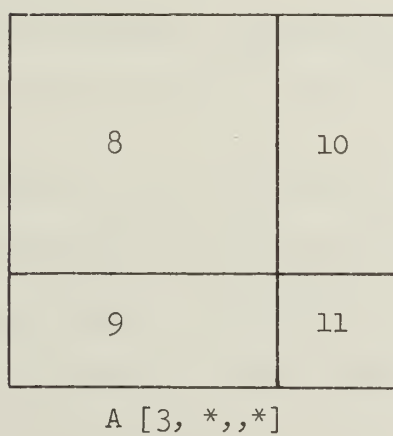
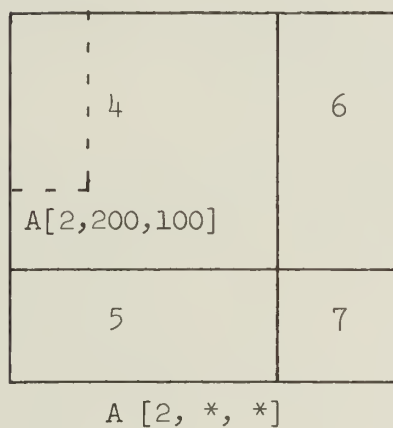
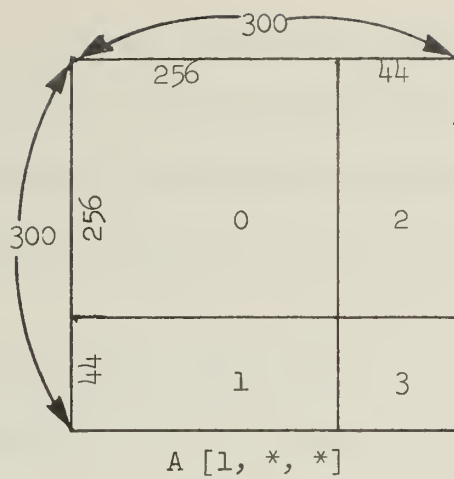


Figure 2. Partitioning for Array A [1:3, 1:300, 1:300]

Figure 2 illustrates the partitioning of a three-dimensional array into 3 subarrays, each with 4 blocks for a total of 12 blocks.

The reasons for using two-dimensional segments are as follows:

- (a) It is necessary to have large segments in order to reduce the number of I/O requests. It is especially necessary that all PEM's be interrupted evenly when I/O requests are being processed [1], i.e., a block should be formed so that it has elements across all PEM's (if possible) when it has been read into memory.
- (b) On the other hand, operations on arrays may always be performed in terms of subarrays; e.g., one can write

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} + \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} (A_{11} + B_{11}) & (A_{12} + B_{12}) \\ (A_{21} + B_{21}) & (A_{22} + B_{22}) \end{bmatrix}$$

for the addition of two matrices A and B provided that submatrices A_{ij} and B_{ij} have the same order.

These two observations suggest the use of two-dimensional segments, i.e., blocks. Another advantage of having two-dimensional segments is that either a row or a column of a segment can be accessed, which is an advantage over linear (row or column) storage. The partitioning of an array into blocks is independent of the mapping function used; i.e., for a SKEWED array skewing is applied to each block after partitioning.

One possible disadvantage which could be claimed against block segmentation is that of difficulty in indexing across blocks. This may, however, be remedied by again utilizing the submatrix concept in matrix operations, e.g., instead of writing:

FOR (I) SIM ([1,2,...,512]) DO

FOR (J) SIM ([1,2,...,512]) DO

BEGIN

$C[I,J] \leftarrow 0;$

FOR (K) SEQ([1,2,...,512]) DO

$C[I,J] \leftarrow C[I,J] + A[I,K] \times B[K,J]$

END

write:

____ (1)

FOR (M) SEQ ([1,2]) DO

FOR (N) SEQ ([1,2]) DO

BEGIN

$[M,N] C \leftarrow 0;$

FOR (L) SEQ ([1,2]) DO

$[M,N] C \leftarrow [M,N] C + [M,L] A \times [L,N] B$

END

____ (2)

where $[M,N]C$ is computed using code (1), but with indices now varying from 1 to 256 instead of 1 to 512, i.e.

```

FOR (I) SIM ([1,2,...,256]) DO
FOR (J) SIM ([1,2,...,256]) DO
    BEGIN
        [M,N] C [I,J] ← 0;
        FOR (K) SEQ ([1,2,...,256]) DO
            [M,N] C [I,J] ← [M,N] C [I,J]
                + [M,L] A [I,K] × [L,N] B [K,J]
        END
    END

```

It should be emphasized that the generation of codes corresponding to (2) above from (1) is the compiler's responsibility and programs written in TRANQUIL do not have to contain explicit provisions for segmentation. For the detailed algorithm for generating code (2) from code (1), the reader is referred to [8,9].

The sizes of the blocks obtained by the array partitioning fall into four categories:

- (a) 256×256
- (b) $n \times 256$ $n < 256$
- (c) $256 \times m$ $m < 256$
- (d) $m \times n$ $m, n < 256$

For the sake of discussion we call these SQUARE, HBLOCK, VBLOCK, and SBLOCK, respectively. As was mentioned before, it is important to form a block which has 256 columns, so that when I/O is being

processed all PEM's will be interrupted evenly. Thus, small blocks belonging to the same array should, if possible, be packed together to form a larger block. The details will be discussed later.

(ii) Predictive information

All array operations and data transfers between ILLIAC IV disk and PE memory are done in terms of these blocks. Blocking facilitates the use of arrays which are larger than the total PE memory. All data is normally stored on the 10^9 bit (approximately 30 memory loads) ILLIAC IV disk, and blocks are only brought into PE memory (at a transfer rate of 5×10^8 bits/second or 8 ms/block) as required. The disk rotation time is 40 ms. The TRANQUIL compiler will automatically generate block transfer I/O requests, which are handled by the operating system, to make it possible to write a TRANQUIL program which includes no explicit I/O statements. All data are initially stored on the disk in the same way they are used on ILLIAC IV; e.g., if a skewed array is necessary on ILLIAC IV, then the array is also stored in a skewed fashion on the disk before the program which uses it is initiated by the operating system.

It is generally agreed that both preplanned and dynamic storage allocation have advantages for certain types of problems. Preplanned allocation should work best with more regularly predictable problems, whereas dynamic storage allocation can best cope with problems whose flow and storage requirements are highly data dependent and therefore unpredictable. In the case of the TRANQUIL compiler it has been further recognized that the methods are not mutually exclusive and can work well in combination. A prescheduled

sequence of I/O requests, for example, can be generated locally; i.e., in a TRANQUIL statement involving large arrays as in

$$A \leftarrow B \times C$$

where A, B, and C are arrays of size 1024×1024 .

On the other hand, a dynamic storage allocation scheme is necessary globally; i.e., between TRANQUIL statements, and this should consider transfers of control. The first TRANQUIL compiler will adopt a simple strategy; i.e., a block on demand system utilizing the first-in last-out strategy.

The block number for an element $A[i_1, i_2, \dots, i_{n-1}, i_n]$ of an array $A[1:M_1, 1:M_2, \dots, 1:M_n]$ is given by

$$\left(\dots \left((i_1 - 1) M_2 + (i_2 - 1) M_3 + \dots (i_{n-2} - 1) M_{n-1} + (i_{n-1} - 1) M_n + i_n \right) \right)$$

$$\cdot \left[\frac{M_n + 255}{256} \right] + \left[\frac{i_n}{256} \right] \left[\frac{M_{n-1} + 255}{256} \right] + \left[\frac{i_{n-1}}{256} \right]$$

For example, the block number for the element $A[2, 200, 100]$ in Figure 2 is

$$((2-1) \left[\frac{300 + 255}{256} \right] + \left[\frac{100}{256} \right]) \left[\frac{300 + 255}{256} \right] + \left[\frac{100}{256} \right] = 4$$

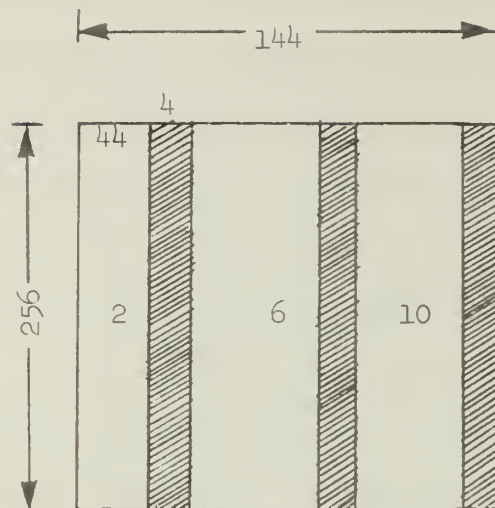
For each block thus established an entry is made in BASETB. This, however, does not necessarily imply that a segment is established for each block. The blocks smaller than 256×256 , i.e., HBLOCKS, VBLOCKS, and SBLOCKS may be packed together (they are then called subblocks) to form a larger block (called a superblock). It should be noted that the terms HBLOCK, VBLOCK, SBLOCK and SQUARE only refer to the size of a block as previously defined. Thus it is possible, for example, to talk about a superblock which is a HBLOCK.

The entry in BASETB gives an absolute base address (the address of the upper left-most element) of the block, if the block is in memory, or the address relative to the base address of the corresponding superblock and a pointer to the BASETB entry of that superblock in the case of a subblock. The above addresses are each specified by a PEM word number (X) together with a PEM number (Y). Thus if $(x, y), (X, Y)$ are the relative address of a subblock and base address of a corresponding superblock, respectively, then the base address of the subblock is $(x + X, y + Y)$.

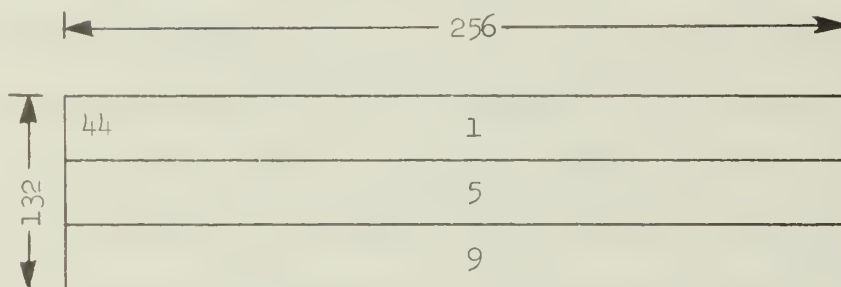
As an example, in Figure 2 there are:

3 SQUAREs	(blocks 0, 4 and 8),
3 HBLOCKS	(blocks 2, 6 and 10),
3 VBLOCKS	(blocks 1, 5 and 9), and
3 SBLOCKS	(blocks 3, 7 and 11).

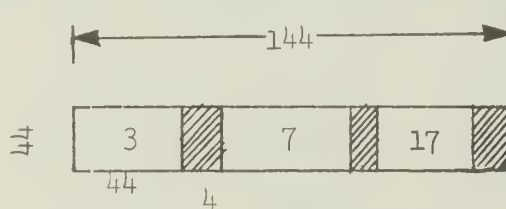
Twelve entries are established in BASETB corresponding to these blocks. The 3 VBLOCKS are packed into a 256×144 superblock (Figure 3).



Packing of VBLOCKS



Packing of HBLOCKS



Packing of SBLOCKS.

Figure 3. Block Packing for Array A[1:3, 1:300, 1:300]

The relative address, in this superblock, of block 10 is (0, 96). The three HBLOCKS are packed into a 132×256 superblock and the three SBLOCKS are stored in a 44×144 superblock as shown.

Further, suppose these blocks are stored in PE memory as shown in Figure 4.

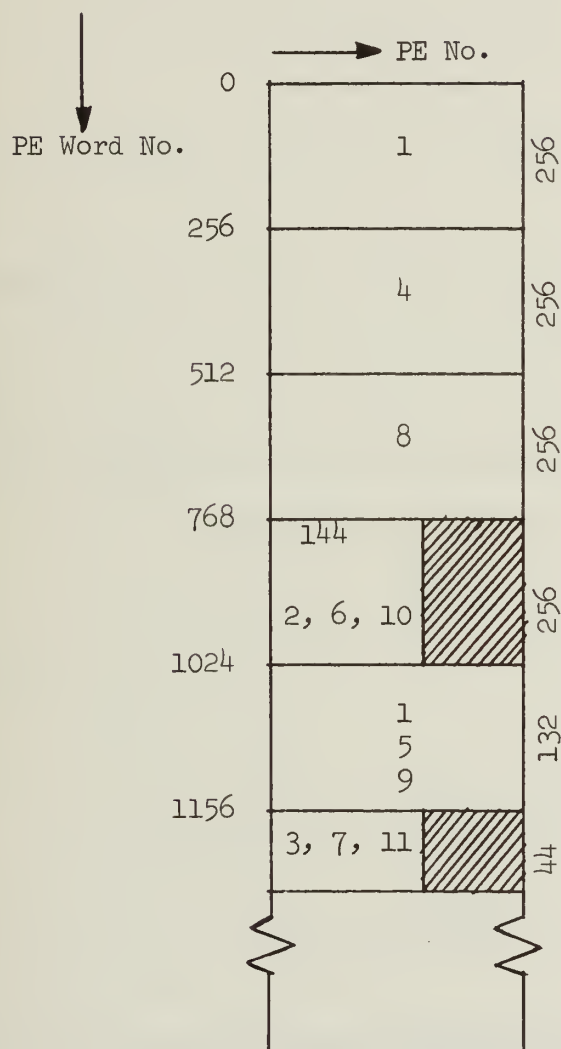


Figure 4. Use of PE Memory for the Blocks Shown in Figures 2 and 3.

Entries for BASETB are made after packing.

P	X	Y	SIZEX	SIZEY
---	---	---	-------	-------

Figure 5. BASETB Format 1

The BASETB entry format for a superblock or SQUARE is shown in Figure 5, where

P indicates that this is an entry for a superblock or SQUARE;

(X,Y) is the absolute base address of this block;

SIZEX is the number of rows of this block;

SIZEY is the number of columns of this block.

C	ORIGIN	COX	COY
---	--------	-----	-----

Figure 6. BASETB Format 2

Figure 6 shows the BASETB entry format for a subblock, where

C indicates that this is an entry for a subblock;

(COX, COY) is the relative address of the subblock.

For example, the BASETB entries for blocks 2 and 10 in Figure 4 are:

2	P	768	0	256	144
10	C	2	0	96	

Figure 7. Examples of BASETB Entries

Thus, the absolute base address for block 10 is given by (X_{10}, Y_{10}) where

$$\begin{aligned} X_{10} &= \text{BASETB}[10] \cdot \text{COX} + \text{BASETB}[\text{BASETB}[10] \cdot \text{ORIGIN}] \cdot X \\ &= 0 + 768 = 768 \end{aligned}$$

and

$$\begin{aligned} Y_{10} &= \text{BASETB}[10] \cdot \text{COY} + \text{BASETB}[\text{BASETB}[10] \cdot \text{ORIGIN}] \cdot Y \\ &= 96 + 0 = 96 \end{aligned}$$

The procedure above generates, for example, 100 subarrays and 100 entries in BASETB for an array

$$A[1:10, 1:10, 1:3, 1:5]$$

in spite of the fact that the subarrays might be packed and generate only one superblock.

An array

$A[\#1:10, \##1:10, *1:3, **1:5]$, on the other hand, generates only 1 entry in BASETB. This is because * and ** together with

and ## force the compiler to generate a single subarray of size 30×50 (Figure 8). Since partitioning is done in terms of this subarray, only one block is generated, making only one entry in BASETB. Thus if *'s and #'s are used effectively, more efficient code can be generated.

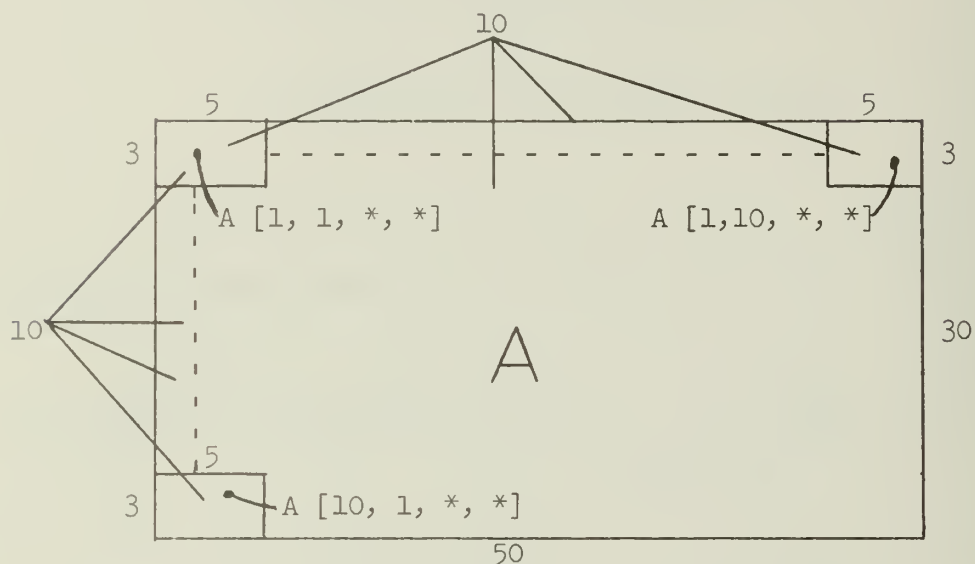


Figure 8. One Subarray Generated from Array
 $A[\#1:10, \##1:10, *1:3, **1:5]$

Should VBLOCKS or SBLOCKS result from partitioning, their sizes will be modified so that they have $\left\lceil \frac{(n + 7)}{8} \right\rceil$ columns, where n is the number of columns of the original block. This is done in order that each block be stored beginning at the p -th PEM, where $p \bmod 8 = 0$, to make efficient use of the 8 word CU fetch capabilities.

Blocks will be packed after the above mentioned modification has been made. Packing is done in two stages. First, packing is done for the same kind of blocks (e.g., VBLOCKS are packed by themselves) belonging to the same array. For example, VBLOCKS are arranged side by side until the number of columns of the resultant superblock reaches 256 . A similar procedure is used for packing HBLOCKS, i.e., they are stacked so that the resultant superblock has up to 256 rows. In the case of SBLOCK packing, first they are treated as VBLOCKS (e.g., they are arranged side by side). Further, if the resultant superblock is a HBLOCK, then HBLOCK packing is done. It should be noted that even after the packing it is possible to have residual blocks which can not be packed, or a superblock which is either VBLOCK, or SBLOCK. These are the objectives of further packing, which is discussed in the following section. HBLOCKS will not be considered in further packing and will be treated as SQUARES.

A detailed flowchart for partitioning and packing is given in Appendix C.

3.3 Address Calculation

The effective address of any array element is established by computing its block number, which was discussed in the previous section, and a relative address within that block. The computation of the relative address varies from one mapping function to another. Here only standard mapping functions are discussed.

For an element $A [i_1, i_2, \dots, i_{n-1}, i_n]$ of an array $A [1:M_1, \dots, 1:M_n]$, the relative PE number and the relative PE word address of the element in the specified block are given as follows, where x will denote PEM address and y will denote PEM number:

(i) STRAIGHT array

$$x = (i_{n-1} - 1) \bmod 256$$

$$y = (i_n - 1) \bmod 256$$

(ii) SKEWED array

$$x = (i_{n-1} - 1) \bmod 256$$

$$y = (i_{n-1} + i_n - 2) \bmod 256$$

For example, consider again the array in Figure 2. The block number for the element $A[2, 200, 100]$ was 4. If the array is skewed, the PE number relative to the base address of this block is $(220 + 100 - 2) \bmod 256 = 42$ and the relative address in this PEM is 199.

The equation in Section 3.2 can also be written as:

$$\left(\left(\dots \left((i_1 - 1) \cdot M_2 + (i_2 - 1) \right) \cdot M_3 + \dots (i_{n-2} - 1) \right) \right. \\ \left. \cdot M'_n + \left[\frac{i_n}{256} \right] \right) \cdot M'_{n-1} + \left[\frac{i_{n-1}}{256} \right]$$

$$\text{where } M'_i = \left[\frac{M_i + 255}{256} \right] .$$

To obtain the block number using this form requires (n-2) subtractions, (n-1) multiplications and (n-1) additions. This value, in turn, is used to access BASETB to locate an absolute base address of the block. $M_1 \cdot M_2 \dots M_{n-2} \cdot M'_{n-1} \cdot M'_n$ words are required in BASETB for this array, besides n words in DOPETB, which contains the bounds for each subscript position. Upon finding the absolute base address, the relative address of an element in the block is calculated using one of the two sets of equations given above, if a standard mapping function is specified. Both equations require one or two subtractions or additions and a shift operation. In practice, a PEM word address is calculated in the PE which requires it, and is used as an index value for the PEX. Thus, the PE index value together with the CU index value, which is an absolute base address of the block, is used to locate the element of the array. In most cases some or all of the elements in a row or column are used simultaneously. In the case of column operation, for example, each PE can simultaneously compute the relative address (index value) which it will require.

It should be noted that to compute the PEM number and the PEM word address of an element, no information on the array size is necessary; i.e., no reference to DOPETB is necessary. On the linear memory space, however, all dimensional information is required to locate the memory cell corresponding to an array element [10].

3.4 Storage Allocation

The storage allocation procedure is a separate package which is independent of the other parts of the compiler, such as the block packing procedure. The compiler can request any amount of memory space; i.e., SQUARE, HBLOCK, VBLOCK or SBLOCK, and free it at any time. The storage allocation procedure keeps track of memory usage, and returns appropriate memory space on request, or frees it.

In allocating memory space for a block a linked space list called I4MEMORY, which keeps count only of the number of rows of memory which have been used, is utilized. If a HBLOCK of size $m \times 256$ (or a SQUARE) is to be stored, the list is searched to locate the smallest space which has at least m (or 256) adjacent rows and the block is stored there. In the case of VBLOCK ($256 \times m$) 256 rows of PE memory may be allocated and a sublist corresponding to a 256×256 block of storage called VLIST is established. A VLIST records use of columns (PEMs) in a particular 256×256 block. In the case of a SBLOCK, again a 256×256 block may be allocated with associated list SLIST. This block is divided into 4×8 subblocks. SLIST consists of a 64×32 bit boolean array (Figure 9) in which each bit represents the use, or otherwise, of each 4×8 subblock.

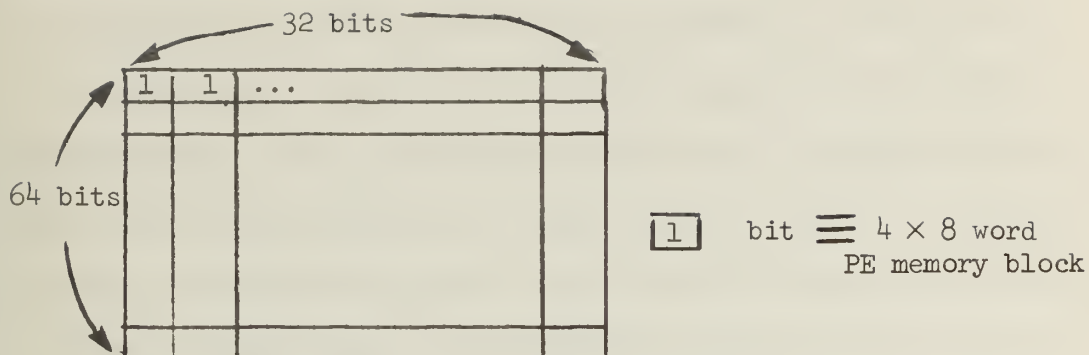


Figure 9. SLIST

The above is the overall picture of the storage allocation. However, in transferring data between ILLIAC IV disk and PE memory, the initial PE memory address is restricted to certain PEM's and the block of data transferred can have only 16, 32, 64, 128 or 256 columns. For example, a 64 column block transfer must start in one of PEM numbers 0, 64, 128 or 192. Thus, adjustment must be made of the size of VBLOCKS and SBLOCKS so that they have one of the above mentioned numbers of columns. This implies that a certain amount of storage is wasted. For instance, a VBLOCK of size 256×72 is made up to a block of size 256×128 , wasting 256×56 words. To avoid this further packing is introduced which is applied before PE memory is assigned to them(i.e., before requesting memory space to the storage allocator).

First a bit array which is similar to SLIST is prepared. Upon entering a TRANQUIL program block, arrays are partitioned and packed as discussed in Section 3.2. If any residual VBLOCKS or SBLOCKS resulted, then they are now formed into a superblock and entries are made in the bit array. A superblock of size $m \times 256$ (SQUARE) is made whenever this bit array becomes full, or the program exits from the TRANQUIL block. This method may still introduce uneconomical packing. However, such wasted space can not be used during that program block anyway.

The strategy that has been chosen might involve considerable bookkeeping, but it minimizes storage fragmentation and reduces the frequency of storage allocation and I/O requests. Details of the algorithm are given in Appendix C.

4. FURTHER DISCUSSION

There are several parts of the TRANQUIL language still to be implemented. One of these is the PEM storage allocation statement. This will be implemented using a macro generator which is now being written. For instance, consider the array B, the storage allocation statement for which appears in Chapter 2. After this statement in a TRANQUIL program, whenever the array B is used its subscript expression is replaced by the expression appearing in the storage allocation statement.

Data management, such as partitioning of arrays and packing of blocks, has been discussed and it has been shown that effective address calculation for elements of partitioned arrays can be computed efficiently. However, no provision has yet been made for a proper way to communicate with the ILLIAC IV operating system. Obviously, for efficient transfer of data from the disk to/from the PEM the format (mapping and partitioning) of the data on the disk must be in the form required in PEM. Hence the TRANQUIL compiler must communicate the appropriate information to the operating system which, when it obtains file name, program name and external format information, will cause the data to be partitioned and mapped in the format required by the associated TRANQUIL program.

Also relating to input/output is the strategy of better resource allocation. One direction which is now being studied is to make a model of a program and interchange statements so that the resultant program is computationally equivalent to the original one,

yet faster in execution speed. The idea is to reorganize statements to minimize input/output of data blocks. Such high optimization will be especially effective for lengthy production type programs.

Finally, TRANQUIL is by no means a completely satisfactory problem oriented language in the sense that users are not free from the burden of choosing data representation, e.g., mapping functions. What is needed is a language which Balzer calls a dataless programming language [11]. He states:

"The independence (of a processing to data representation) will allow the programmer (1) to disregard, while specifying the program, the details of data processing, memory space requirements, and matching of data representation to the processing done on it; and (2) to handle them instead, during the data declaration phase. The problem of data representation can be left until this programming has been completed; thus, a more rational decision can be made concerning an optimal representation. Because of this separation, the programmer should be able to think through his problem better."

5. CONCLUSION

Array mapping functions and array partitioning for the TRANQUIL language have been discussed in this paper. It is obvious that for array type operations SKEWED storage will often provide the best result in terms of computational speed because either a row or a column of a SKEWED array can be accessed simultaneously by all PEs. Also, since the use of a TWS system [12] makes the TRANQUIL compiler more readily modifiable, it is feasible to incorporate new mapping functions in TRANQUIL relatively easily. It should be emphasized that the addressing mechanism for arrays is rather simple in spite of the complicated partitioning. Actually, to locate an element in a block, only a few additions/subtractions and some shift operations are required. Thus, the partitioning mechanism has no adverse effect on the computational speed of compiled codes.

Finally the partitioning and blocking of data, together with automatic (computer generated) input/output of data blocks to/from disk, enables the user to write programs using data arrays (up to 10^9 bits) which are larger than the ILLIAC IV memory.

2. Variable Declaration

2.1 Syntax

<variable declaration> ::= <attribute> <variable list>

<attribute> ::= BOOLEAN | REAL | REALS |
REALD | INTEGER | INTEGERS |
INTEGERL | BYTE8 | BYTE16

<variable list> ::= <variable list>, <variable> |

<variable>

<variable> ::= <identifier>

2.2 Examples

INTEGER I, J

REAL X, Y

2.3 Semantics

Variable declarations serve to declare certain identifiers to represent simple variables of a given type. Each attribute corresponds to a specific word format in ILLIAC IV:

<u>BOOLEAN</u>	64 bit word (only the least significant bit is meaningful)
<u>REAL</u>	64 bit floating point
<u>REALS</u>	32 bit floating point
<u>REALD</u>	128 bit (double precision) floating point
<u>INTEGER</u>	48 bit fixed point
<u>INTEGERS</u>	24 bit fixed point
<u>INTEGERL</u>	64 bit fixed point (no sign)

<u>BYTE 8</u>	8 bit fixed point (no sign)
<u>BYTE16</u>	16 bit fixed point (no sign)
<u>COMPLEX</u>	Simple variables further specified by this attribute have complex numbers as values.

3. Array Declaration

3.1 Syntax

```

<array declaration> ::= <mapping function> ARRAY <array list> |
    <attribute> <mapping function> ARRAY <array list> |
    ARRAY (<PEM area>) <array list> |
    <attribute> ARRAY (<PEM area>) <array list>

<mapping function> ::= STRAIGHT | SKEWED | SKEWED PACKED |
    CHECKER | <empty>

<mapping procedure name> ::= <identifier>

<array list> ::= <array segment> | <array list>, <array segment>

<array segment> ::= <array identifier> [<bound list>] |
    <array identifier>, <array segment>

<bound list> ::= <bound> | <bound>, <bound list>

<bound> ::= <arrangement> <bound pair> | <arrangement> <limit>

<arrangement> ::= * | ** | # | ## | <empty>

<bound pair> ::= <lower bound> : <upper bound>

<lower bound> ::= <arithmetic expression>

<upper bound> ::= <arithmetic expression>

<limit> ::= <arithmetic expression>

<PEM area> ::= <identifier>

```

3.2 Examples

REAL SKEWED ARRAY A[1:5, 1:10], B[5, 10]

ARRAY AR, BR[**1:80, *1:256]

INTEGER ARRAY AI, BI[##4, **64, *128]

3.3 Semantics

An array declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables and gives the dimensions of the arrays, the bounds of the subscripts, the types of the mapping functions, and the type of the variable.

3.3.1 Subscripts Bounds

As in PL1, the subscript bounds can be given in either ALGOL form or FORTRAN form. Simultaneous use of both forms in one bound list is, however, forbidden; e.g., [1:5, 10] is illegal.

3.3.2 Dimensions

Up to eight dimensions are allowed in arrays.

3.3.3 Arrangement (Also see Chapter 1.)

Generally arrays are stored in PE memory by subarrays, which are made up from the last two dimensions; e.g., three 64×128 subarrays are to be formed for an array A[3, 64, 128]. Arrangement declarations serve to change the rule. One asterisk *, together with ** placed in front of a bound indicates that a subarray is formed by those dimensions. For example, A[5, **256, *128] will generate five

subarrays of size 128×256 . Since subarrays are stored in PE memory as they are, i.e., a $m \times n$ subarray occupies m PEM words in n PEM's, the arrangement declaration may be used to introduce a better memory usage. Also, $*$ forces data corresponding to that subscript to be stored in one PEM, and $**$ indicates that the data for that subscript is stored across PEM's. Thus, a vector $A[50]$ can be stored in one PEM by declaring as $A[*50]$, or across PEM's by $A[**50]$. One sharp $\#$ and two sharps $\#\#$ similarly indicate how to arrange subarrays in PE memory. One sharp indicates the direction of increasing PEM word address and $\#\#$ indicates across PEM's. For example an array $A[\#5, \#\#4, *32, **64]$ will introduce twenty 32×64 subarrays, arranged in five rows of 4 subarrays thus making up a 160×256 block (Figure A1).

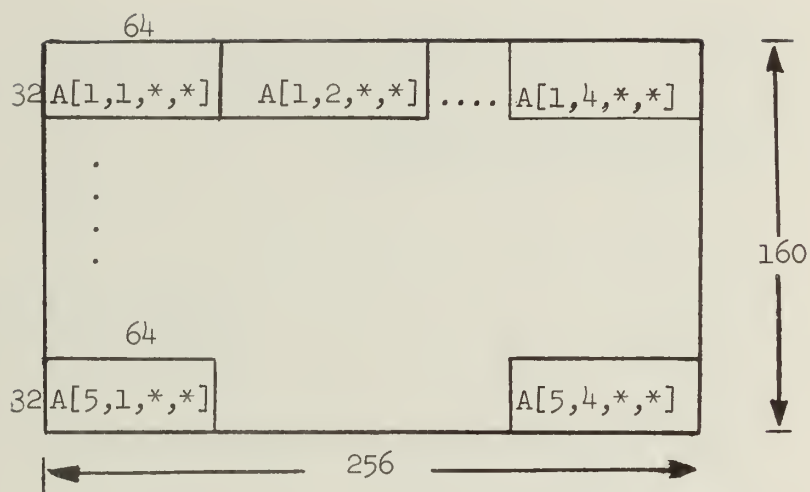


Figure A1. Subarrays for an Array $A[\#5, \#\#4, *32, **64]$

The five combinations of arrangement markers given in Table A1 are legal, where the numbers indicate the number of allowable appearances in a single declaration.

	*	**	#	##
1st combination	n-times			
2nd	1	1		
3rd	1	1	1	
4th	1	1		1
5th	1	1	1	1

Table A1. Combinations of Arrangement Declaration Markers

3.3.4 Mapping Function

The default mapping function is SKewed PACKED. In the case of a user-specified mapping function, a corresponding PEM assignment declaration must be in effect at the time the array declaration is processed.

3.3.5 PEM Area

If a PEM assignment declaration is used to define a special mapping function, then a corresponding PEM area name must appear in the array declaration.

4. PEM Reserve Declaration

4.1 Syntax

`<PEM reserve declaration> ::= PEMEMORY <PEM area name>
[<word size>, <PEM size>]`

`<PEM area name> ::= <identifier>`

`<word size> ::= <unsigned integer>`

`<PEM size> ::= <unsigned integer>`

4.2 Example

PEMEMORY PEMEM [10, 256]

4.3 Semantics

PEM reserve declarations serve to reserve a certain amount of virtual memory, allowing the programmer to store arrays there in any fashion. The integer used for PEM size should not be greater than 256. Should more space be needed, more than one area may

be reserved. The declaration should appear before the area is used in a PEM assignment declaration. The reserved area will be released upon exit from the block in which it was declared, as usual.

4.3.1 Word Size and PEM Size

Both word and PEM are understood to be numbered starting from 0 and increasing in increments of 1.

5. PEM Assignment Declaration

5.1 Syntax

```

<PEM assignment declaration> ::= PEM <PEM assignment block>

<PEM assignment block> ::= <PEM assignment construction> |
                               BEGIN <list of PEM assignments> END

<list of PEM assignments> ::= <PEM assignment construction> |
                               <PEM assignment construction>; <list of PEM assignments>

<PEM assignment construction> ::= <PEM assignment statement> |
                                   <set assignment statement> |
                                   <PEM for statement>

<PEM for statement> ::= <PEM for clause> <PEM assignment block>

<PEM for clause> ::= FOR (<set variable list>) SIM
                    (<set name list>) DO

<PEM assignment statement> ::= <PE memory> ← <array name>

<PE memory> ::= <PE area> [<word index>, <PEM index>]

<word index> ::= <unsigned integer> | <variable>

<PEM index> ::= <unsigned integer> | <variable>

<array name> ::= <array identifier> [<subscript list>]

```

$\langle \text{subscript list} \rangle ::= \langle \text{subscript} \rangle | \langle \text{subscript list} \rangle, \langle \text{subscript} \rangle$
 $\langle \text{subscript} \rangle ::= \langle \text{arithmetic expression} \rangle$

5.2 Examples

```

PEM FOR (I,J) SIM ([1,2,...,256] × [1,2,...,256]) DO
    PEMEM[I,J] ← ARRAY [J,I]

PEM BEGIN
    PEMEM [1,1] ← ARRAY [1,1];
    PEMEM [256,1] ← ARRAY [1,256];
    FOR (I) SIM ([2,3,...,255]) DO
        PEMEM [I,1] ← ARRAY [I,1]

    END

```

5.3 Semantics

PEM assignment declarations serve to store arrays into a reserved PEM area in the way specified, i.e., declare a new mapping function.

5.3.1 PEM Area

PEM areas must be reserved before they are actually used by PEM assignment declarations.

5.3.2 Variables

All variables appearing in PEM assignment declarations besides PEM area names and array names need not be declared and they are understood to be local to the declaration.

5.3.3 Sets

All sets appearing in a PEM assignment declaration may not be dynamic; i.e., parameters can never be passed to this declaration from outside.

5.3.4 PEM Assignment Statement

PEM PEMEM [I,J] \leftarrow AR [K,L]

causes the (K,L) element of an array AR to be stored in the I-th row and the J-th column of PEMEM.

5.4 Further Example

BEGIN

PEMEMORY PEMEM [256,256];

PEM

FOR (I,J) SIM ([1,2,...,256] \times [1,2,...,256]) DO

PEMEM [I,J] \leftarrow ARAY [J,I];

REAL ARRAY (PEMEM) ARAY [256,256];

.
.
.

END

In the above example a 256×256 array ARAY is stored in PE memory in such a way that a column of ARAY is across PEMs and a row of ARAY is in a single PEM.

APPENDIX B

TABLES

The following is the list of tables used in the TRANQUIL compiler to take care of declarations and memory allocation.

(i) Tables used in both Pass 1 and Pass 2

IDTAB contains the information on each identifier declared in a program; e.g., type, a pointer to a corresponding DOPETB entry if an identifier is an array.

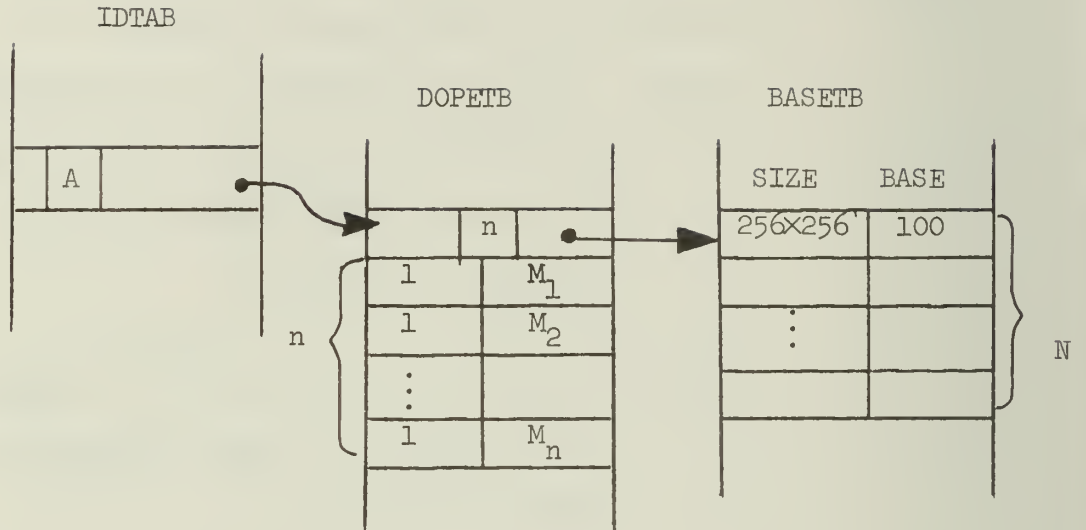
DOPETB contains the information necessary to reference arrays; e.g., size of each dimension, the number of dimensions.

(ii) Tables in Pass 2

BASETB contains the descriptor for each block resulting from an array partitioning; e.g., size of a block, base address for a block.

These tables are linked as shown in Figure B1. The number of blocks N in BASETB is determined by:

$$N = M_1 \times M_2 \times \dots \times \left[\frac{M_{n-1} + 255}{256} \right] \times \left[\frac{M_n + 255}{256} \right]$$



n = number of dimensions

N = number of blocks

Figure B1. Entries and Linkage of Tables for

$$A[1:M_1, 1:M_2, \dots, 1:M_n]$$

APPENDIX C

ARRAY PARTITIONING

AND

PACKING FLOWCHARTS

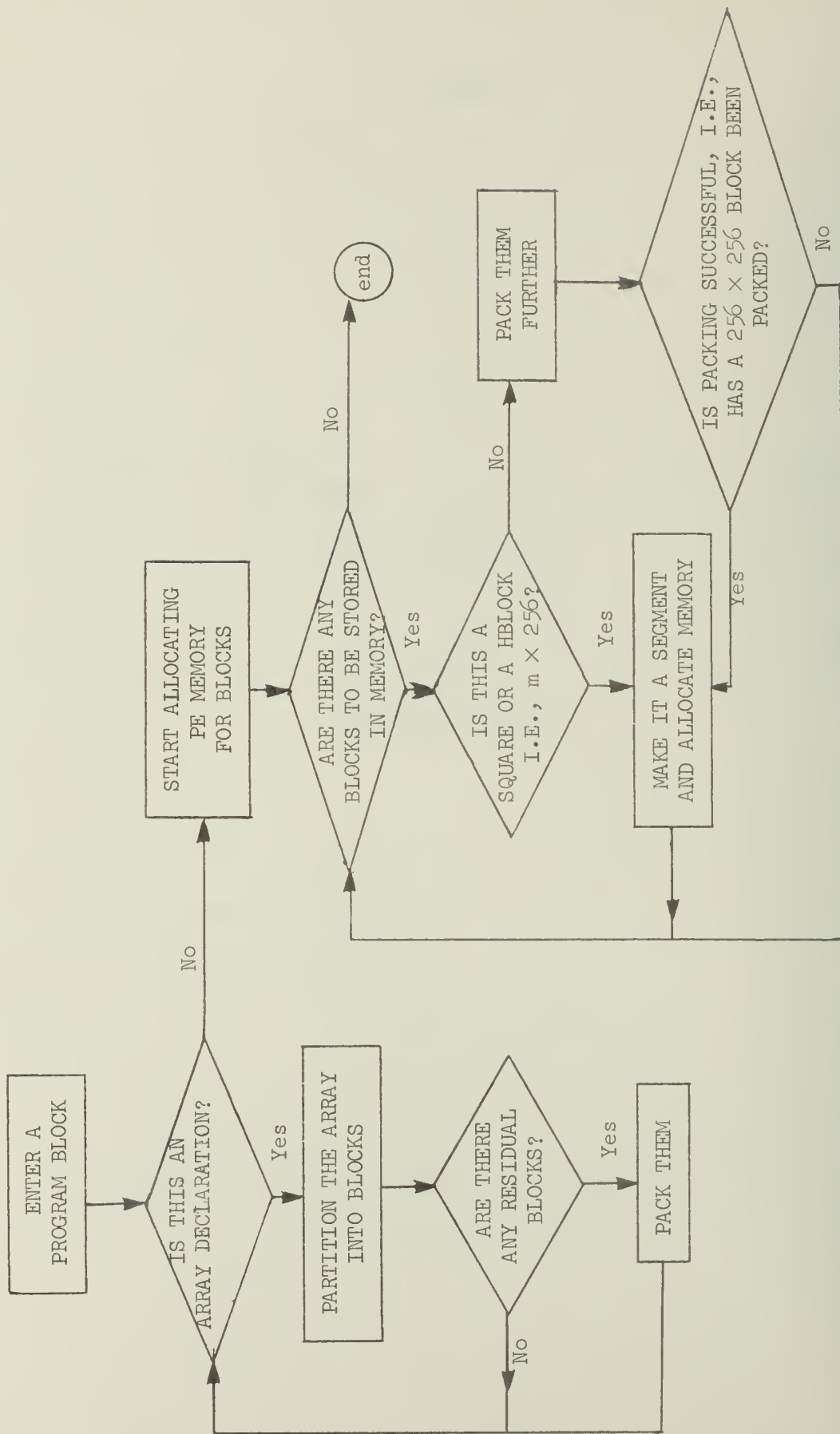


Figure C1 (Part 1) . Pass 2 Program Block Entry and Block Exit Flowcharts for Array Declarations.

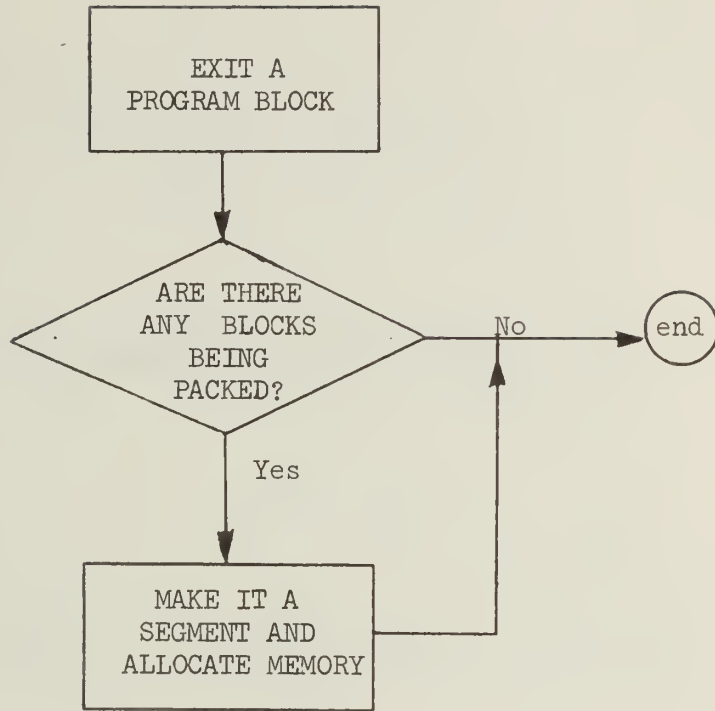


Figure C1 (Part 2). Pass 2 Program Block Entry and Block Exit Flowcharts for Array Declarations.

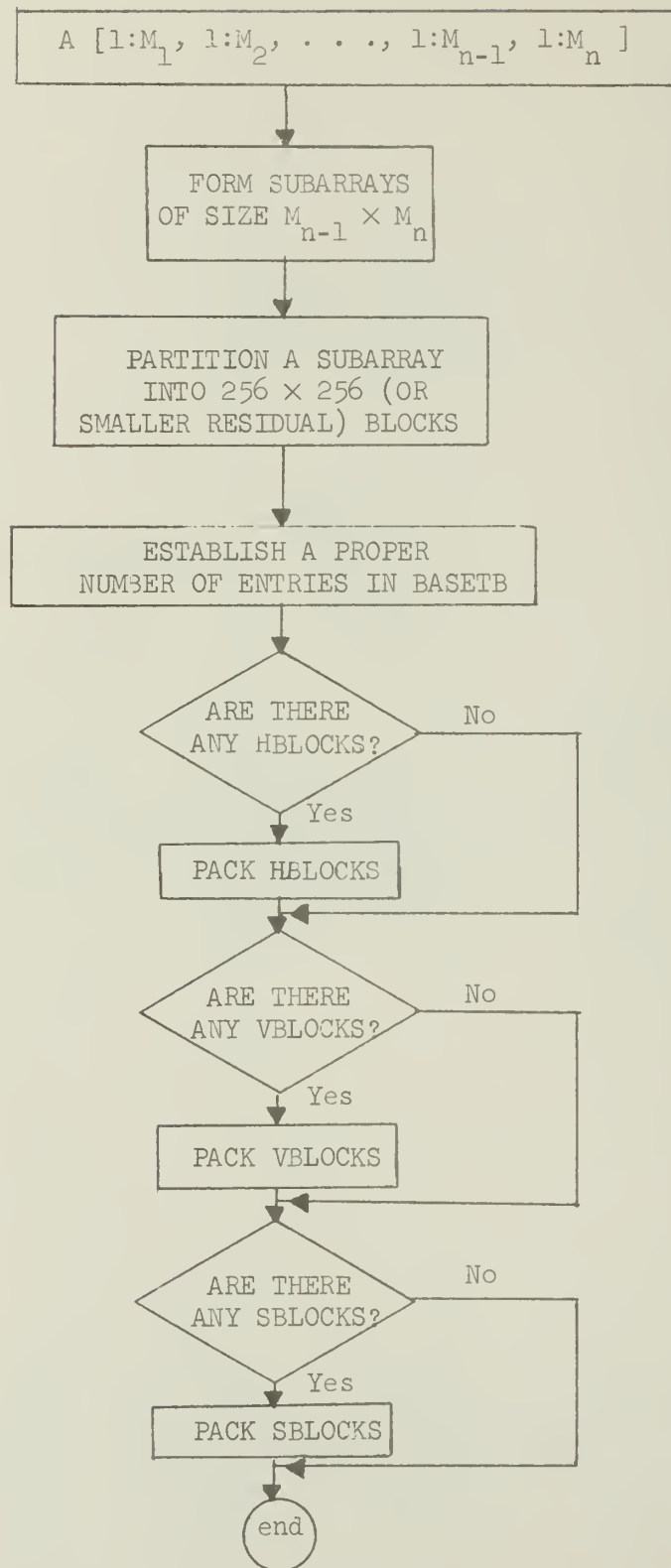


Figure C2. Array Partitioning Flowchart

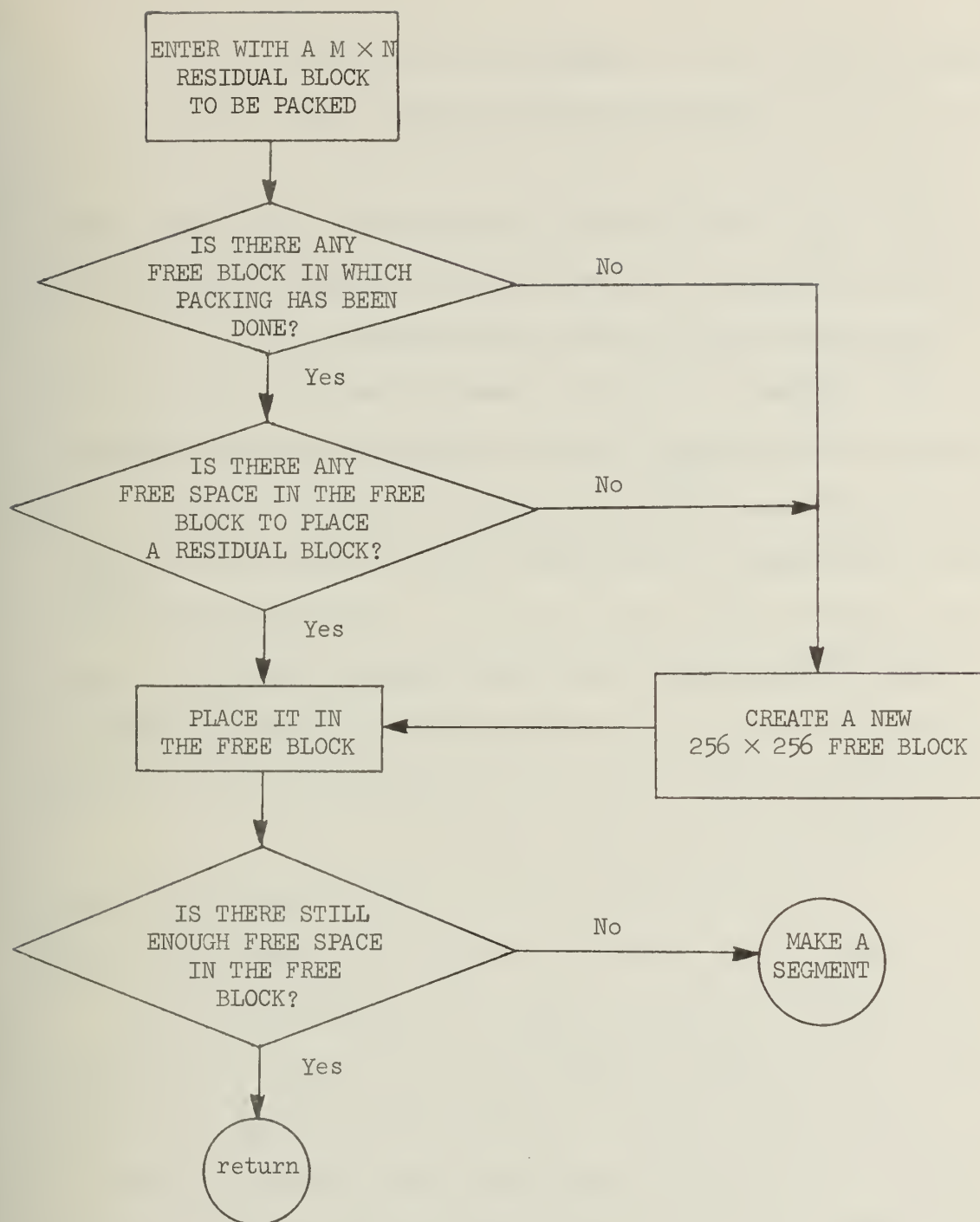


Figure C3. Residual Block Packing Flowchart

APPENDIX D

STORAGE ALLOCATION PACKAGES

There are six separate procedures for storage allocation. Any one of them can be called at any time. To maintain a record of PE memory usage, three lists or tables are used.

14MEMORY is a dynamic linked list which records the use of PEM rows. Corresponding to any $m \times 256$ PEM memory block assignment there is an element of the list which contains both an origin of this block in PEM, and the number of PEM rows used, i.e., m . The list contains a similar entry (except for 1 bit) for each block of available storage in PEM. Each element also has two pointers, one pointing to that element (free or used) which has the next higher origin, and the other pointing to the element of the same kind (free or used) having the next higher origin.

To allocate memory space for a block of size $256 \times m$, the table VLIST is used. This is essentially a 16 element array, each of whose elements corresponds to usage of 16 adjacent PEMs. This implies that the number of PEMs (i.e., m) can only be a multiple of 16 (actually either one of 16, 32, 64 or 128 is allowed due to the hardware specification). Upon encountering a request for memory space of size $256 \times m$, a check is made for the existence of a current VLIST table. If none exists, then a 256×256 block of PEM is reserved and a VLIST table is created. Then VLIST is searched until $\left\lceil \frac{m}{16} \right\rceil$ adjacent free columns are found, the appropriate PEM is allocated, and the appropriate table entries are flagged.

The SMALLSPACE table is used to allocate space for smaller size blocks. This is a 64×16 bit boolean array in which each bit represents a 4×16 word block of PEM (1 = allocated, 0 = free). Thus SMALLSPACE represents a 256×256 PEM block. This implies that the valid memory block size which can be requested should have numbers of rows and columns which are multiples of 4 and 16, respectively. To find freespace of size $m \times n$, the first $\left\lceil \frac{m}{4} \right\rceil$ rows of SMALLLIST are anded together and searched from left to right (i.e., from 0-th PEM to 255-th PEM) for $\left\lceil \frac{n}{16} \right\rceil$ consecutive 0's. If this is unsuccessful, then the same process is repeated on the next $\left\lceil \frac{m}{4} \right\rceil$ rows and so on.

Figures D1, D2 and D3 show the formats for the above tables and examples of table entries.

(a) I⁴MEMORY entry word

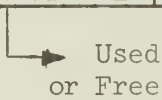
F or U	ORIGIN	POINTER 1	POINTER 2	SIZE	POINTER To BASETB
--------	--------	-----------	-----------	------	----------------------

F = Free

U = Used

(b) VLIST entry format

AVAILABLE	SIZE	POINTER TO BASETB
-----------	------	-------------------


 Used
or Free

(c) Structure of VLIST

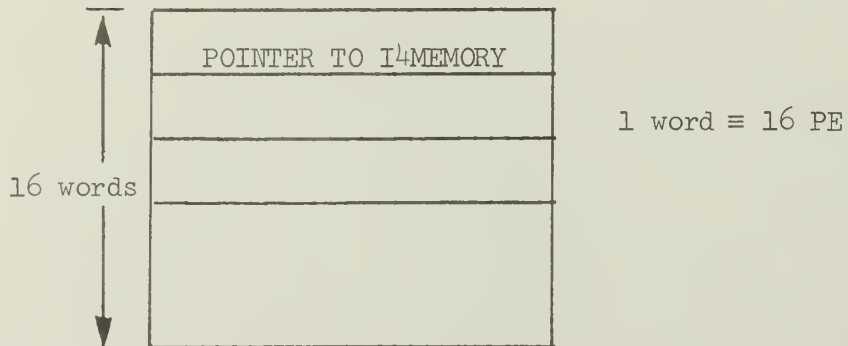


Figure D1. Table and List Entry Formats

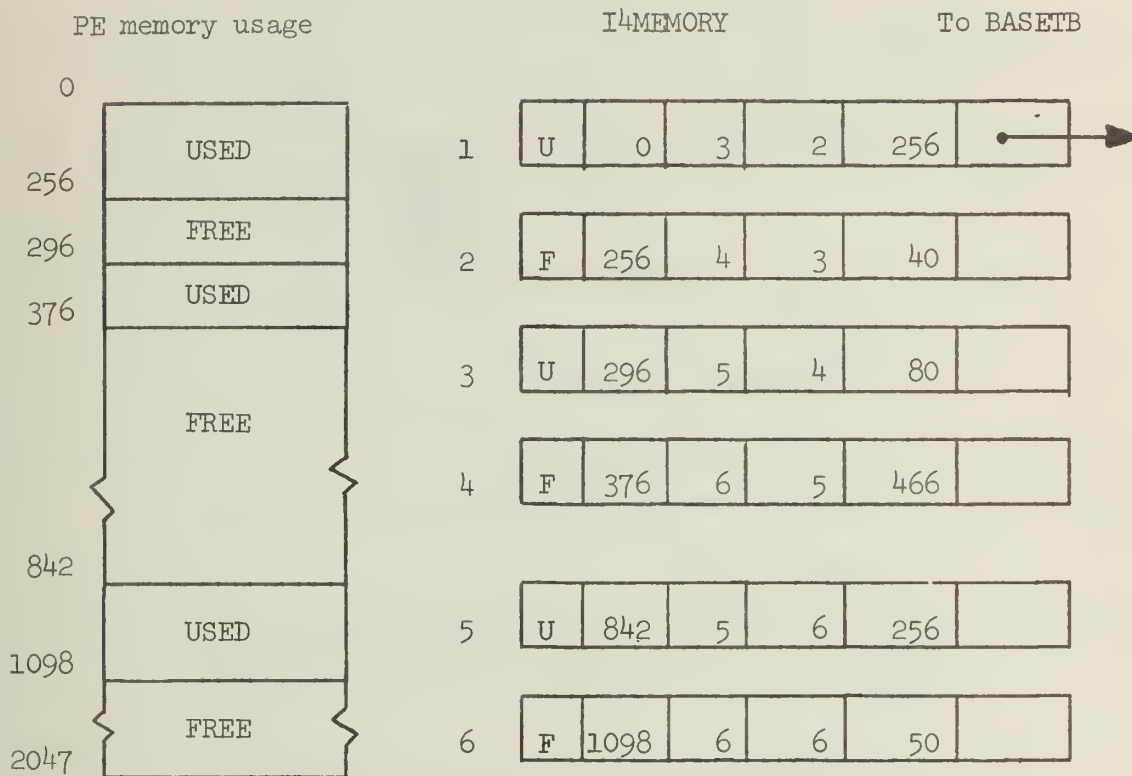


Figure D2. Example of an Entry for I4MEMORY

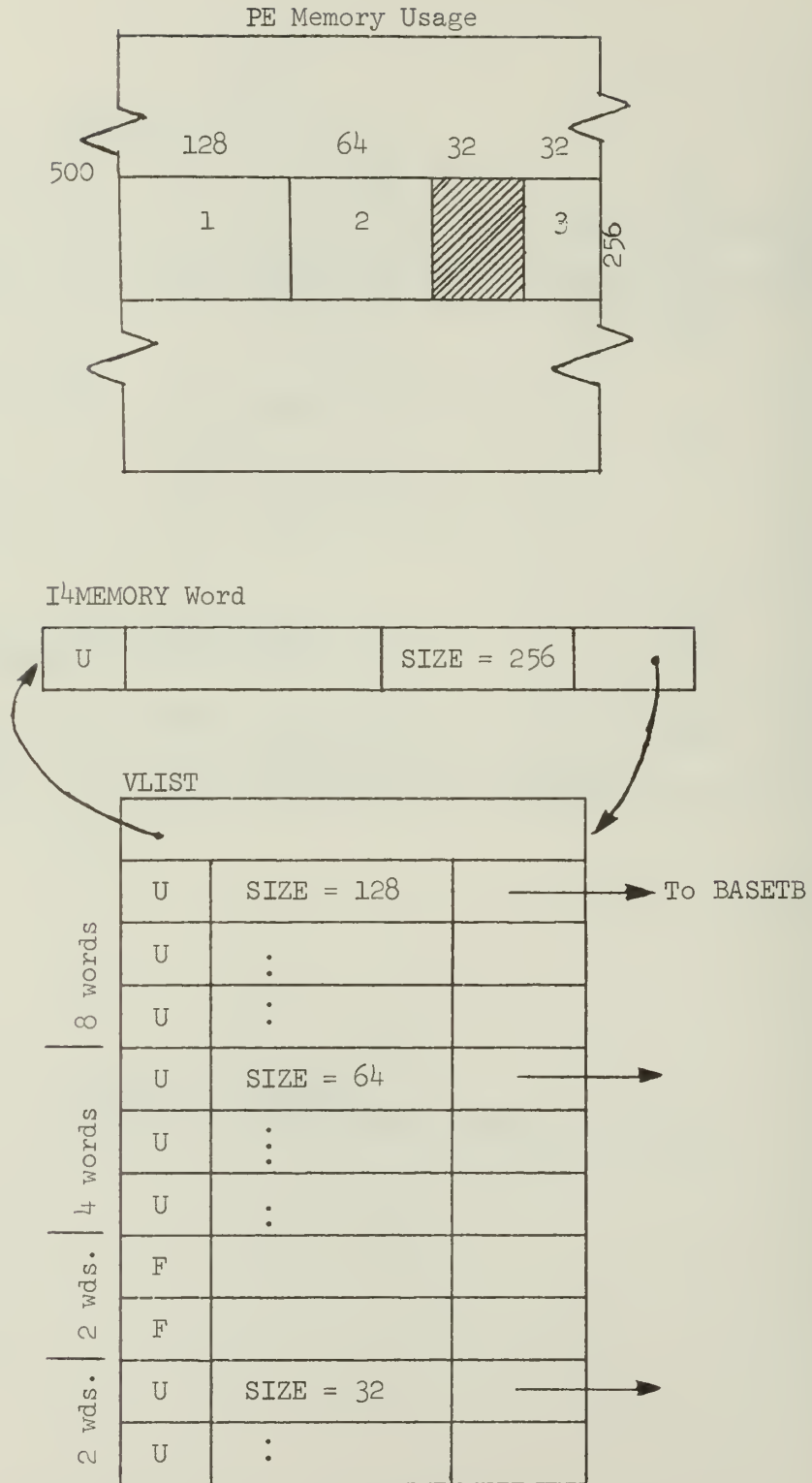


Figure D3: Example of an Entry for VLIST

LIST OF REFERENCES

- [1] Barnes, G. H., et al, "The ILLIAC IV Computer", IEEE Transactions on Computers, C-17, 8 (August, 1968), pp. 746-757.
- [2] Kuck, D. J., "ILLIAC IV Software and Application Programming", IEEE Transactions on Computers, C-17, 8 (August, 1968), pp. 758-770.
- [3] Iverson, K. E., "A Programming Language", John Wiley & Sons, Inc., New York (1962).
- [4] Knuth, D. E., "The Art of Computer Programming", Vol. 1, Addison-Wesley (1968).
- [5] Knowles, M., et al, "Matrix Operations on ILLIAC IV", Department of Computer Science, University of Illinois, Urbana, Illinois, ILLIAC IV Document No. 118 (March, 1967).
- [6] Benokraitis, V., "Alternate Storage Methods for Two-Dimensional Hydrodynamics Calculations", Department of Computer Science, University of Illinois, Urbana, Illinois, ILLIAC IV Document No. 190 (May, 1968).
- [7] Randell, B. and Kuehner, J. C., "Dynamic Storage Allocation Systems", Comm. ACM, 11, 5 (May, 1968), pp. 297-306.
- [8] Wilhelmson, R. B., "Control Statement Syntax and Semantics of a Language for Parallel Processors", (M.S. Thesis), Department of Computer Science, University of Illinois, Urbana, Illinois, (January, 1969).
- [9] Budnik, Paul P., "TRANQUIL Arithmetic", (M.S. Thesis), Department of Computer Science, University of Illinois, Urbana, Illinois, (January, 1969).
- [10] Hellerman, H., "Addressing Multidimensional Arrays", Comm. ACM, 5, 4 (April, 1962), pp. 205-207.
- [11] Balzer, R. M., "Dataless Programming", Proc. FJCC, (1967), pp. 535-543.
- [12] Northcote, R. S., "The Structure and Use of a Compiler-Compiler System", Proc. Third Australian Computer Conference, (May, 1966), pp. 339-344.
- [13] Backus, J. W., "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference", Proc. Int. Conf. Inf. Proc., UNESCO, Paris, France, (June, 1959).

- [14] Naur, P., et al., "Revised Report on the Algorithmic Language ALGOL 60", Comm. ACM, 6 (January, 1963), pp. 1-17.

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

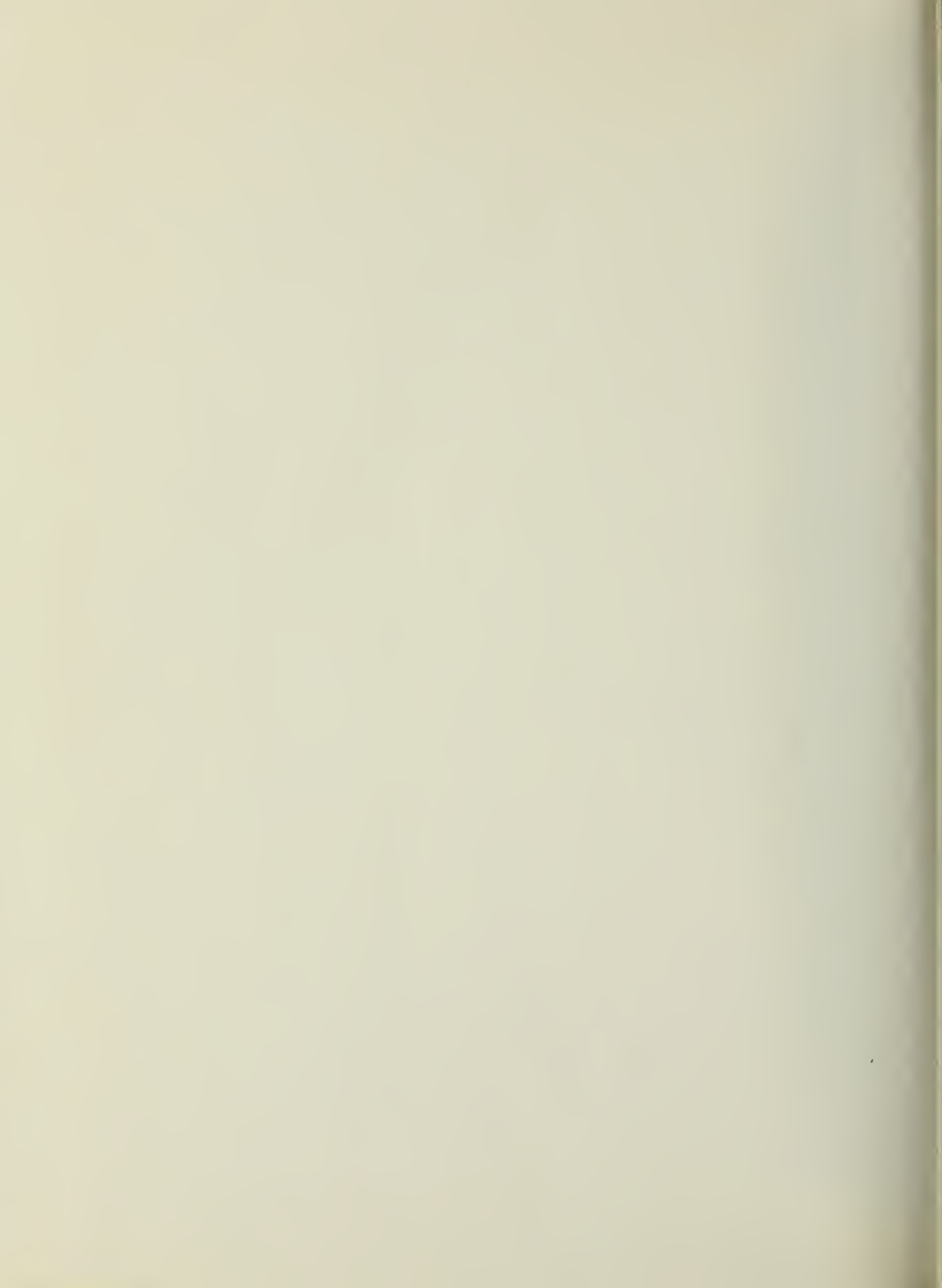
(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Department of Computer Science University of Illinois Urbana, Illinois 61801		2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	
		2b. GROUP	
3. REPORT TITLE STORAGE ALLOCATION ALGORITHMS IN THE TRANQUIL COMPILER			
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) Research Report			
5. AUTHOR(S) (First name, middle initial, last name) Yoichi Muraoka			
6. REPORT DATE January 13, 1969		7a. TOTAL NO. OF PAGES 61	7b. NO. OF REFS 14
8a. CONTRACT OR GRANT NO. 46-26-15-305		8b. ORIGINATOR'S REPORT NUMBER(S) DCS Report No. 297	
b. PROJECT NO. USAF 30(602)4144			
c.		9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	
d.			
10. DISTRIBUTION STATEMENT Qualified requesters may obtain copies of this report from DCS.			
11. SUPPLEMENTARY NOTES NONE		12. SPONSORING MILITARY ACTIVITY Rome Air Development Center Griffiss Air Force Base Rome, New York 13440	

13. ABSTRACT TRANQUIL is a language for describing algorithms in terms of parallel constructs. Its compiler is now being implemented for the parallel array computer ILLIAC IV. This paper discusses a particular part of the implementation; namely, the problem of storage allocation for arrays.
--

UNCLASSIFIED
Security Classification

MAY 15 1973





UNIVERSITY OF ILLINOIS-URBANA



3 0112 045402051